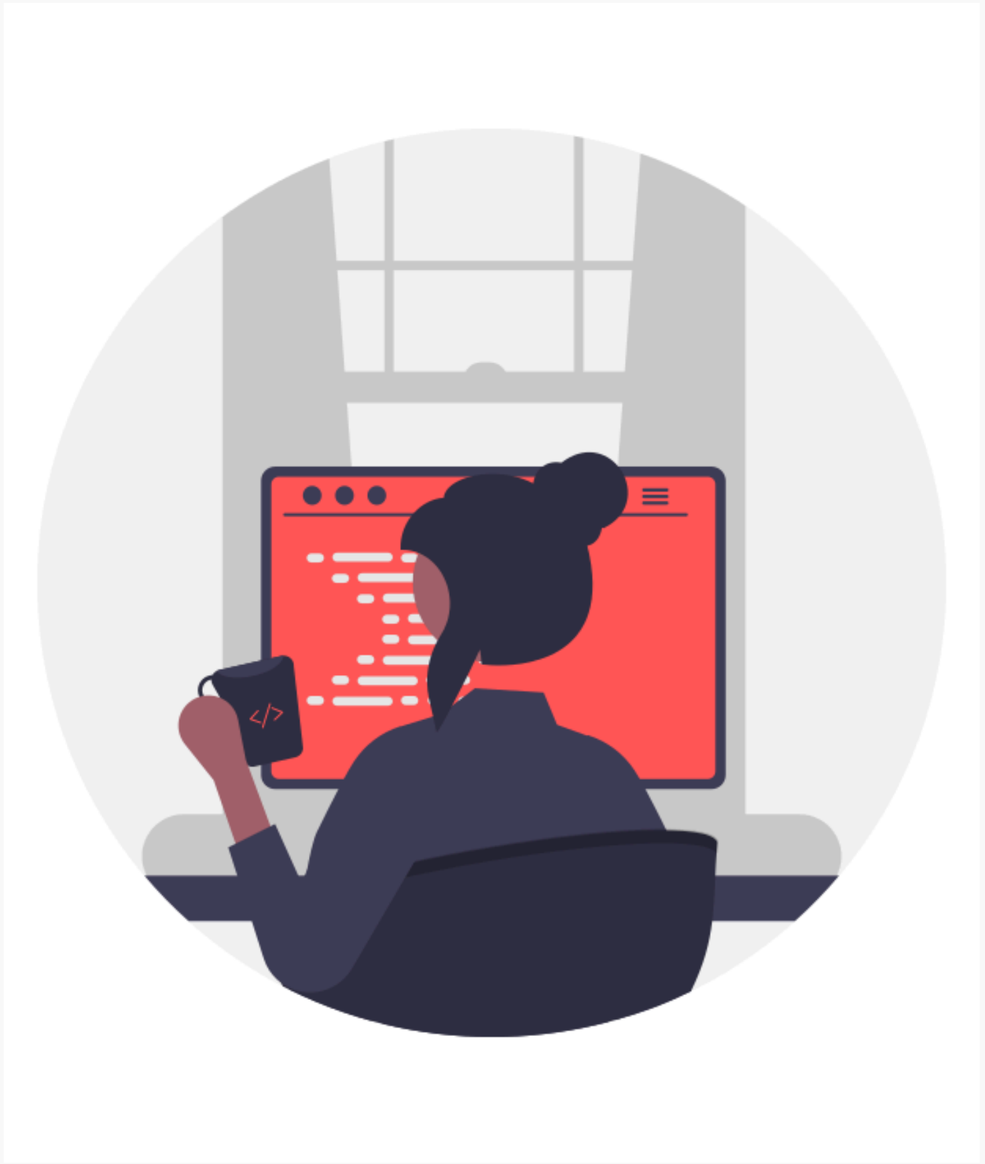
Writing Clean Code in Python

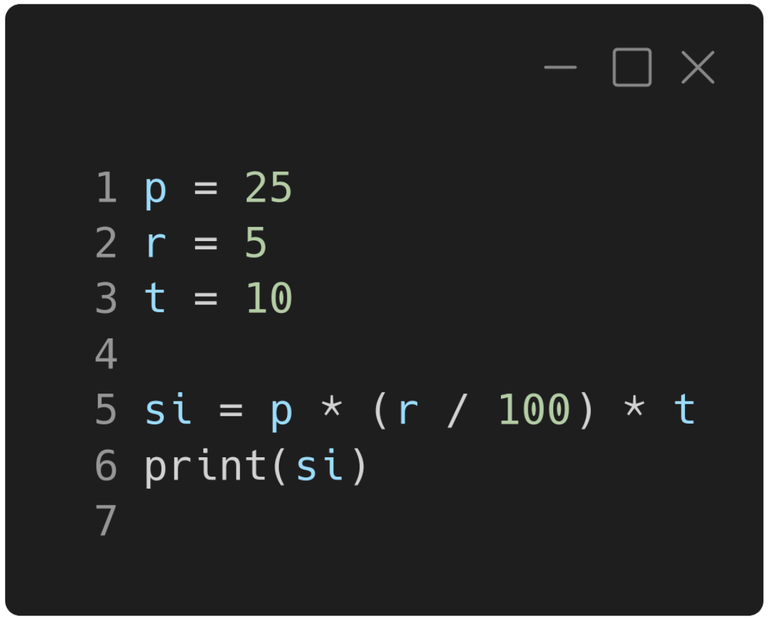
Cognext Python Team



What is clean code?

- Readability
- Well-documented
- Consistent styles and conventions
- Modularised
- Simple
- Maintainable
- Testable

Readability



```
1 p = 25
2 r = 5
3 t = 10
4
5 si = p * (r / 100) * t
6 print(si)
7
```

BAD CODE!

Readability

Use meaningful variable names

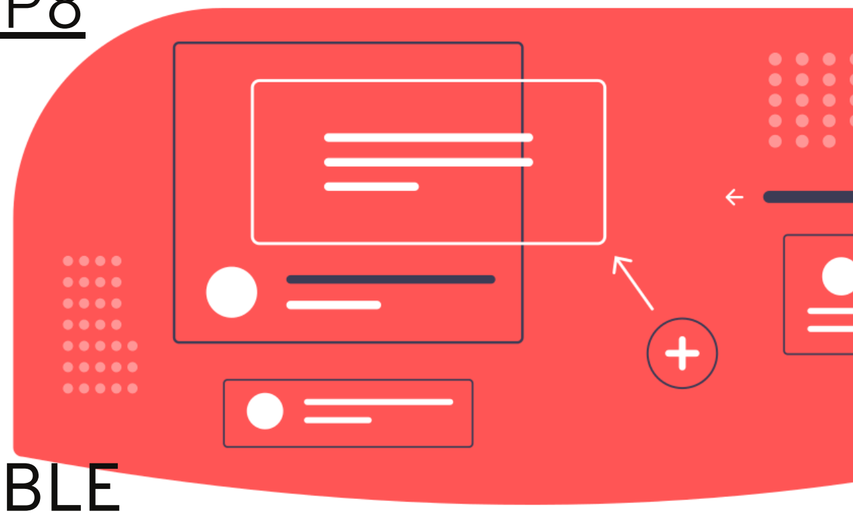
```
principal = 25
annual_rate_in_percent = 5
time_in_years = 10

simple_interest = principal * (annual_rate_in_percent / 100) * time_in_years
print(simple_interest)

```

Readability

PEP8



- Style guidelines for Python code
  - variables, functions: **snake_case**
  - global_variables, constants: **GLOBAL_VARIABLE**
  - classes: **CamelCase**, **HTTPConnection**
  - private methods, attributes: **__name**

pep8.org

Readability

- Introduced conventions for type annotations
- Dynamically typed
- Makes code more understandable
- Catch bugs early

https://peps.python.org/pep-0484/

Readability

```
1 def getSimpleInterest(principal, annual_interest_rate_in_percent, time_in_years):
2   simple_interest = principal * (annual_rate_in_percent / 100) * time_in_years
3   return simple_interest
4
5 simple_interest = get_simple_interst(25, 5, 10)
6 print(simple_interest)
7
```

BAD CODE!

Readability

```python
def get_simple_interest(
  principal: float,
  annual_interest_rate_in_percent: float,
  time_in_years: int
) -> float:
  return principal * (annual_rate_in_percent / 100) * time_in_years


simple_interest = get_simple_interest(25, 5, 10)
print(simple_interest)

```

Documentation

PEP257

- Conventions for docstrings
- numpy style
    - clear, rich-text support.
    - numpy, pandas, scipy etc.
- Explain purpose, parameters, return values, potential errors/warnings.
- Developers and end-users

https://numpydoc.readthedocs.io/en/latest/format.html

https://peps.python.org/pep-0257/

```python
def get_simple_interest(
    principal: float,
    annual_interest_rate_in_percent: float,
    time_in_years: int
) -> float:
    """
    Calculate the simple interest.

    Parameters:
    principal (float): The principal amount.
    annual_interest_rate_in_percent (float): The annual interest rate in percentage.
    time_in_years (int): The time period in years.

    Returns:
    float: The computed simple interest.

    Example:
    >>> get_simple_interest(25, 5, 10)
    12.5
    """
    return principal * (annual_interest_rate_in_percent / 100) * time_in_years


simple_interest = get_simple_interest(25, 5, 10)
print(simple_interest)

```

Simplicity

PEP20 - Zen of Python

- `import this`
- Nested loops and conditions ⇒ comprehensions and generators
- Use built-in functions and standard libraries
- Break down tasks into small, reusable functions.
- Each function should do one thing and do it well - SRP

https://peps.python.org/pep-0020/

Simplicity

```python
list_with_duplicates = [1, 7, 2, 6, 3, 9, 2, 4, 5, 2, 9, 1, 3, 8]
deduplicated_list = []


# using for loop
for item in list_with_duplicates:
  if item not in deduplicated_list:
    deduplicated_list.append(item)


# using comprehensions
deduplicated_list = [item for index, item in enumerate(list_with_duplicates) if item not in list_with_duplicates[:index]]

deduplicated_list = list({item for item in list_with_duplicates})


# using built-in functions
deduplicated_list = list(set(list_with_duplicates))
```

Simplicity

Single Responsibility Principle

```python
#bad
def get_discounted_price(items, discount_rate):
    total = 0
    for item in items:
        total += item
    discounted_total = total * (1 - discount_rate)
    return discounted_total


 #good
def calculate_total(items):
    total = 0
    for item in items:
        total += item
    return total

def apply_discount(total, discount_rate):
    discounted_total = total * (1 - discount_rate)
    return discounted_total

```

- Each function should do one thing and do it well.
- Break down tasks into small, reusable functions.

Modularity

- Breaking code into smaller, reusable components.
- Organise code into functions, classes, and modules.
    - **Maintainability:** Easier to understand, debug, and update.
    - **Scalability:** Facilitates adding new features and extending functionality.
    - **Collaboration:** Enables multiple developers to work on different modules simultaneously.
    - **Testing:** Isolate individual units and test them independently

Modularity

```python
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, item):
        self.items.append(item)

    def calculate_total_price(self):
        return sum(item.price for item in self.items)

```

- Encapsulate similar functions and data into Classes.

Modularity

```python
# file: shopping_cart.py
class ShoppingCart:
    def __init__(self, customer_id):
        self.customer_id = customer_id
        self.items = []

    def add_item(self, item):
        self.items.append(item)

    def calculate_total_price(self):
        return sum(item.price for item in self.items)

class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price

# Additional helper functions can also be included in the module
def apply_discount(total_price, discount):
    return total_price * (1 - discount)

```

- Organise similar classes and helper functions into importable modules.
- __init__.py

Testability

- Ease with which the application can be tested.
- Crucial for ensuring code reliability, maintainability, and scalability.
- Enable faster development cycles.
- Fewer bugs in production.

Testability

Writing testable code

- Separation of concerns: Ensure that each function or class has a single responsibility.
- Dependency injection: Pass dependencies as parameters rather than hardcoding, allowing for easier mocking in tests.
- Minimise side effects: Avoid modifying global state or performing I/O operations within functions, making it simpler to reason about and test the code.

Linters and Formatters

- Linters: Analyse code for potential errors, stylistic inconsistencies, and best practices violations.
- Formatters: Automatically format code according to a defined style guide, ensuring consistent and readable code.
- Enhance code quality and maintainability.
- Enforce coding standards and best practices.
- Flake8, Pylint, Black, autopep8, ruff etc

Ruff for Python

- Linter & formatter in one.
- Written in Rust; extremely fast.
- Supports 800+ rules; replacement for *flake8*, *pydocstyle*, *pycodestyle*, *isort*, *black* etc.
- Configurable through *pyproject.toml*
- Easy integration with vscode

https://docs.astral.sh/ruff/

autodocstring

- Automatically generate docstrings.
- Infers type-hints of input parameters and output
- Easy integration with vscode

https://marketplace.visualstudio.com/items?itemName=njpwerner.autodocstring